Interacting with Multiple Alternatives Generated by Recognition Technologies

Jennifer Mankoff & Gregory D. Abowd
College of Computing & GVU Center
Georgia Tech
Atlanta, GA 30332
E-mail: {jmankoff,abowd}@cc.gatech.edu

Scott Hudson
HCI Institute
Carnegie Mellon University
Pittsburgh, PA 15213
E-mail: hudson@cs.cmu.edu

ABSTRACT
Despite significant advances in recognition technologies in areas such as speech and gesture recognition, our experience tells us that recognition errors and uncertainty are unlikely to disappear. For the foreseeable future, use of recognition based systems will introduce uncertainty into the input process. If interactive systems are going to work robustly with recognition-based input, it will be necessary to consider uncertainty as a normal part of input handling rather than considering it to be an anomaly or an exceptional condition. This paper considers techniques for explicit treatment of input uncertainty in user interfaces. In particular, it considers a general class of techniques for the display of, and interaction with, multiple alternatives generated by recognition technologies. Augmentation of the typical event-handling infrastructure is discussed, as well as an application interface infrastructure which attempts to minimize the impact of uncertainty on the application. A prototype system that embodies this infrastructure is also considered.

KEYWORDS: recognition systems, inputs with uncertainty, multiple alternative displays, error handling, toolkits

INTRODUCTION
Recognition-based input is becoming more commonplace. Humans communicate with each other using handwriting, speech, and other natural modes and they have an intuitive understanding of how to use those communication modes. In the case of human-computer communication, handwriting and speech are desirable because they can be used in settings where the keyboard and mouse are inaccessible such as mobile computing and by people with disabilities.

Recognition systems also come with a serious disadvantage — they sometimes incorrectly interpret human input. If recognition systems make incorrect choices, correcting them can slow potentially fast input methods such as speech from around 120 words per minute (wpm) down to 20-30 wpm, half the speed of typing [22]. For slower input methods, such as handwriting, this slowdown can make the system almost unusable for lengthy input. There is extensive work in trying to reduce the number of errors made by these technologies, but experience suggests that the uncertainty introduced by recognizers is unlikely to go away entirely [21].

If uncertainty is being introduced into the interface, It behooves user interface designers not to ignore the uncertainty but to provide explicit mechanisms for dealing with it effectively. We have surveyed previous work in user interface and system-level techniques for handling these errors [16]. We uncovered two common strategies. One is repeating the input in a different modality. For example, Suhm found that it was more accurate to correct speech using a pen than with speech [22]. This is because the sorts of errors that occur in speech input are orthogonal to those that occur in pen input. Another option is to use a more accurate repetition modality such as a soft keyboard.

The second strategy is to present the user with multiple alternatives interpretations, from which a correct one is chosen. Recognizers generate interpretations from input such as speech, handwriting, drawings, or gestures. The potential interpretations may include anything which is the result of recognition, including commands, ASCII text, or drawings. There are a variety of interfaces for selecting an interpretation. Figure 1 shows one example, a menu. Table 1 describes other examples, drawn from our literature survey. We chose to focus our work on this strategy because the variety and complexity of user interface issues which need to be supported make it a more interesting and difficult problem for which to provide reusable support.

This paper addresses the user interface infrastructure needs for generalized support of displaying and interacting with multiple alternatives. We have implemented a variety of interactive components (what we will call “interactors”) including several of the interaction techniques uncovered in our survey. This

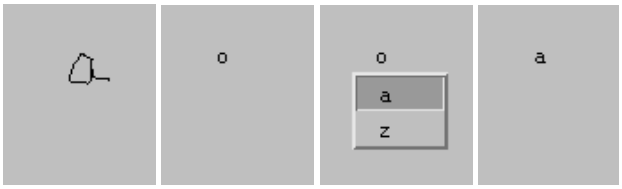

Figure 1: Correcting a misrecognized letter. The user draws an "*a*", which is recognized as an "o". Because this is wrong, the user clicks on the "o", which causes a menu of other alternatives to appear. The user selects "a" from the menu and the error is corrected.

interactor hierarchy can easily be subclassed in order to support additional strategies.

For effective support of the display of multiple alternatives, it is necessary to extend the standard event-based infrastructure to handle *ambiguous events*, input events that have more than one plausible interpretation. This simplifies the implementation of the multiple alternative display, and the use of recognizers. At the same time, we want to hide the ambiguity from the application as much as possible. Our goal is to put up an "ambiguity firewall" between the application and the ambiguity generated by the recognizer. From the application developer's perspective, ambiguous event handling should differ as little as possible from standard event handling. We require only one application-level extension: Applications must separate feedback about potential interpretations of input from the action performed once a correct interpretation has been chosen.

The next section discusses in more detail the variety of multiple alternative displays found in our survey. We then discuss the relationship between a multiple alternative display (abbreviated *MAD*) interactor and the application, including the high-level design choices which make it possible for us to implement the variety found in the survey. The following section presents our architecture, which makes the simplicity of the *MAD* and application design possible. We then demonstrate the architecture by showing how the multiple alternative displays in Figures 1 and 5 were implemented for an example of pen-based input recognition. Our current prototype demonstrations concentrate on pen-based inputs, but these techniques should be applicable to the wider range of recognition-based inputs, such as speech and video. We then discuss the limitations of our approach and future work.

BACKGROUND

We are concerned primarily with speech, pen, and keyboard-based input. These inputs may be translated by various recognition-based technologies into commands, ASCII words, spoken words, letters, or drawings. We surveyed previous research and applications that use recognition-based technologies in order to determine what error handling techniques are currently in use [16]. The complete survey discusses several aspects of error handling including error discovery, testing, toolkit-level support, and interface techniques. Interface techniques include repetition (including redoing, rephrasing, switching to less ambiguous or orthogonally ambiguous input methods, and the use of natural human correction strategies such as overwriting) and displaying multiple alternatives.

In the area of multiple alternative displays, our survey uncovered variations in alternative *layout*, display *instantiation*, whether or not the original input (or other additional *context*) is displayed, *interaction* techniques for choosing an alternative, and the *feedback* used to represent what an alternative looked like. For example, the Newton MessagePad™ uses a menu layout, which is instantiated when the user double clicks on a word. The original handwritten input is displayed at the bottom of the menu and an alternative is selected by clicking the mouse on the choice. Alternatives are represented in this menu as text (words).

Although this work focuses on systems that have graphical output, researchers in audio-only systems have used multiple alternative displays as well. For example, Brennan & Hulteen use natural language to "display" multiple alternatives [3], and Marx & Schmandt use an audio menu in his work [17]. Table 1 gives an overview of some commercial and research systems with graphical output that demonstrates different multiple alternative displays.

Layout: Variations in layout include a pie menu [14], a linear menu [2, 5, 19], or floating around a central location [7], in location that the selected alternative will appear [13], or in a grid [1, 8, 23]. Another variation is to display only the top choice (while supporting interactions that involve other choices) [7].

Instantiation: Variations in when the display is originated include: on a click [2] or other user request such as pause [14] or spoken command [5]; based on an automatic assessment of ambiguity, continuously [1, 8, 18, 19]; as soon as recognition is completed [7, 13]; or on any other input event the designer wishes [23].

System	Layout	Instantiation	Context	Interaction	Feedback
MessagePad™[2]	linear menu	on click	original ink	drag-release	ASCII words
DragonDictate™[5]	linear menu	speech command		speech command	ASCII words
Goldberg and Goodisman [7]	below top choice	on completion		click on choice	ASCII letters
Word Prediction (Alm [1] & Greenberg [8])	bottom of screen (grid)	continuously		click on choice	ASCII words
Word Prediction (Netscape [18])	in place	continuously		returns selects top keystroke, arrow requests more	ASCII words
Marking Menu [14]	pie menu	on pause		flick at choice	commands, ASCII letters
Beautification [13]	in place	on completion	constraints	click on choice	pictures (lines)
Remembrance Agent [19]	bottom of screen, linear menu	continuously	certainty, result excerpts	keystroke command	ASCII sentences
UIDE [23]	grid	on command		click on choice	thumbnails of results

Table 1: The layout, instantiation mode, context, selection, and representation used by systems covered in our survey [16].

Context: A large variety of additional context may be displayed along with the actual alternatives including information about their certainty [19], how they were determined [13], and the original input [2].

Interaction/Selection: Most displays use information about where the mouse was released for selection. For example, Goldberg and Goodisman suggest using a click to select the next most likely alternative even when it is *not* displayed [7]. Many systems automatically select the top choice if the user starts to draw a new stroke [7, 13]. This is called implicit confirmation. In cases where recognition is highly error prone, the opposite is done [1, 8, 19]. This is called implicit contradiction. The system essentially acts as if no recognition had occurred at all.

Feedback: Variations in what is displayed included ASCII text [1, 2, 5, 7, 8, 18, 19], drawings [13], commands [14], icons [23], and mixtures of these types. Table 1 shows what choices the systems covered in our survey made for each of these variables.

Table 1 illustrates these design space requirements. Each system we reference implemented one-off solutions for their particular problem, but as Table 1 makes clear, the same design decisions show up again and again. By providing higher-level support for this design space, we can begin to experiment with new approaches. For example, although continuous display of alternatives has been used in text-based prediction such as Netscape's word prediction and the Remembrance agent [18, 19], to our knowledge it has not been used to display predicted completions of a gesture in progress.

APPLICATION AND INTERACTOR DESIGN

Our primary goal is to provide application designers with reusable interactors for handling errors by displaying multiple alternatives without requiring significant changes to the application itself. In order to be useful, these interactors must be reusable both across applications and across recognizers. This means that a multiple alternative display must work with other standard graphical user interface (GUI) interactors, as Henry *et al.* Discuss [9], and must work with the variety of data types returned by recognizers such as text, commands, and drawings.

In order to support this goal, we separate the interaction and feedback of each individual interpretation from the interaction and feedback with the *MAD* itself. The *MAD* knows nothing about the representation of its constituent alternatives. The *MAD* only handles the instantiation, layout and selection of interpretations. Each interpretation is in charge of displaying itself. These interpretation displays are provided by the application. In fact, they can be any GUI interactor.

The only change required of the application to support this design is the separation of *feedback* about each interpretation from the *action* taken once an interpretation has been selected by the user. The application still provides feedback immediately, but it must wait to act until any ambiguity has been resolved. The application never has to know how many alternatives there are, or gather them together in one place and choose one. It simply returns information about how to provide feedback for any particular alternative when asked. And it acts on the alternative when told that it can.

For example, consider a pen or mouse stroke for which a recognizer returns two alternatives (*delete* and *select).* If the application were to immediately act on both events, it would show some feedback indicating selection and then select the object under the gesture. It would also show some feedback indicating it was deleting the same object

and then delete it. Alternatively, the application designer would have to include some code to gather up all the recognition results, pick one, and then act on that one. In our system, the application provides feedback for indicating the selection and feedback for indicating the deletion. And once the user has chosen, say, *select* with the help of the *MAD¸* the application is told that it can act on *select*. It never has to worry about the fact that there is more than one potential interpretation, and it also doesn't have to worry about executing some incorrect action or worry about trying to figure out which interpretation was selected.

This separation between feedback about interpretations and the *MAD* allows for two interesting extensions. First, each interpretation can support interactions if desired. For example, the feedback interactor for the *delete* event may allow the user to change an argument to that command.

Second, each interpretation can easily display additional application-specific context about itself such as the constraints used to generate a line or the part of speech associated with a word. To do this, the application replaces the interpretation's default feedback interactor when the *feedback()* method is called. This replacement knows how to draw the additional information. No change to the *MAD* itself is required.

AMBIGUOUS EVENT HANDLING ARCHITECTURE

In order to make minimal changes to the application and still keep the design of *MAD*s simple, we had to build a new event-handling infrastructure. Figure 2 shows how the flow of events in our previous example would happen in a standard GUI application that uses a recognizer to turn strokes into commands. This scenario can be generalized to handle other input modalities besides pen. Here the application must decide when to call the recognizer, and then take its response and either pick one interpretation to act on, or create a *MAD*. All of the uncertainty is exposed to the application designer, who must figure out the best way to handle it. This results in error-handling solutions which are not reusable and can make the design of the application more complicated. If the application uses a *MAD¸* that *MAD* only has access to the information that the application designer chooses to give it, which may limit its capabilities.

In contrast, our ambiguous event handling system treats the interpretations returned by the recognizer as events just like a *mouse click* or *key press*. This allows us to deliver interpretations to interested interactors in the same way that standard events are delivered.

Figure 3 shows the flow of events for the same example as Figure 2, but with the addition of our ambiguous event-handling infrastructure. Now the *MAD* can automatically subscribe directly to the event-handling infrastructure in order to be informed about new interpretations. The gesture recognizer no longer has to wait for the application to call it, but can ask the architecture to notify it whenever a stroke is completed. And the application designer does not have to worry about deciding which interpretation is correct —once the user has selected an interpretation, the application will automatically be notified that it can act on that choice.

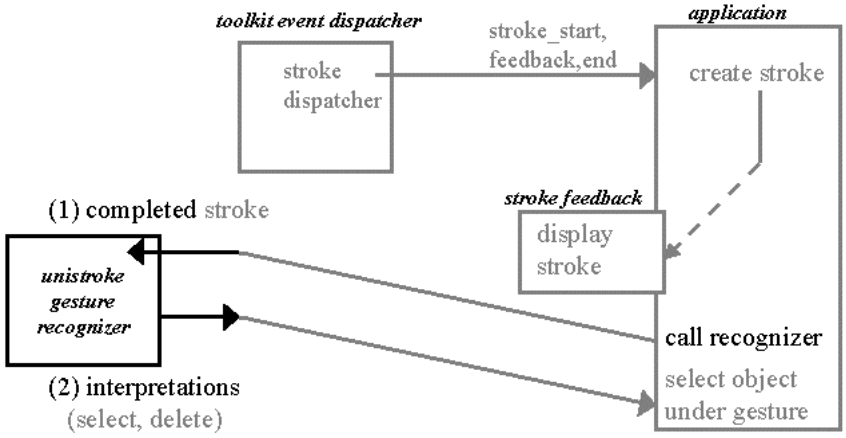

Figure 2: The flow of events in a simple application written with a standard event architecture. Gray indicates components and events present in most standard GUIs. The recognizer is black because it, like the additions in Figure 3, involves the use of and delivery of ambiguous events.

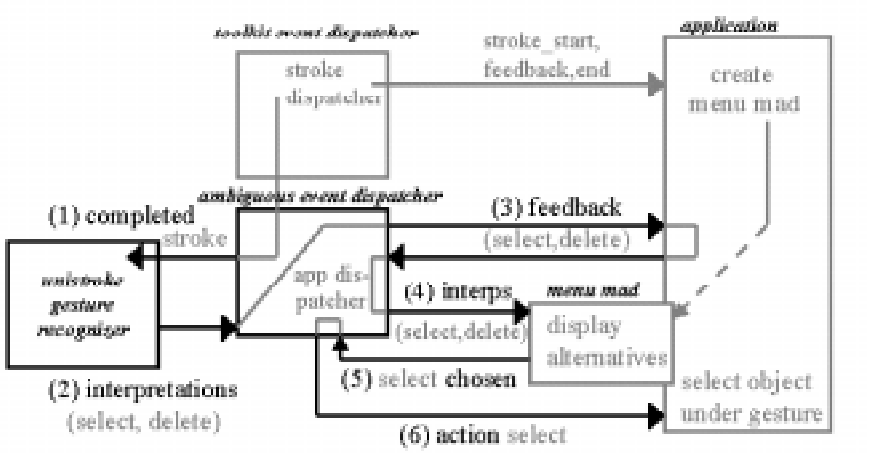

Figure 3: The flow of events in a simple application written with our architecture. Gray indicates components and events present in Figure 2. Stroke events are delivered normally to the application. But whenever a stroke is completed it is also automatically delivered to the recognizer (1), which then returns a set of interpretations (2). The ambiguous dispatcher then requests *feedback_interactors* from the application (3), and passes them along with the interpretations to the menu multiple alternative display (4). When the user selects one, the dispatcher is notified (5) and it tells the application it can now act on that interpretation (6).

Extensions to Standard Event Handling

In order to allow recognized input to be treated just like any other event, we had to make two major extensions to the event handling infrastructure.

ambiguous events: First, we had to create a new class of events which can be recognized. These are referred to as *ambiguous events* because they may have several potential interpretations, only one of which is correct from the user's perspective. Ambiguous events are described in more detail below. Because of this change, we refer to our architecture as an *ambiguous event handling system.*

event life-cycles: In our architecture, events are no longer *consumed* as they are in standard event handling. For example, in a GUI toolkit such as subArctic [6, 10], an interactor returns a *Boolean* value indicating whether or not an event was consumed. If the return value is true, the toolkit has finished with that event, and no other interactor has access to it. If false, the toolkit assumes that interactor did not want or use the event, and it passes the event on to the next interactor. In contrast, interactors which handle *ambiguous events* return an *ambiguous event change* object which indicates what, if any, changes were made to the event. Every change to an event sends it back through the event queue.

Because of this change, we deliver information not only about the creation of ambiguous events, but also about any changes to ambiguous events. For example, interactors, *MAD*s, or recognizers may wish to know when an interpretation has been chosen, when an input event has been recognized, or even about each update to a stroke as the user draws it.

Consider the example in Figure 3. In our system, a stroke is an example of an ambiguous event, since it can be recognized. When a stroke is completed, it enters the event queue and is passed to the recognizer (arrow 1). The recognizer returns its recognized interpretations in an ambiguous event change object (arrow 2). These are sent back through the event queue. This example will be discussed in more detail below, after we define ambiguous events.

Ambiguous Events

In order to better understand how an ambiguous event may change, we need to define it more precisely. An ambiguous event is any event that provides input to a recognizer, hence may have multiple interpretations. An ambiguous event keeps a pointer to a list of its potential interpretations. Figure 4 shows an example of a stroke event with two interpretations (the *delete* command and the *select* command).

Each interpretation has a percentage certainty (generated by the recognizer) [11]. In our example, the *delete* command is 80% certain while the *select* command is only 20% certain.

User interaction with a *MAD* leads to a selection of one interpretation as "correct." While certainty represents the recognizer's best guess of which interpretation is correct, we store information about the user's opinion in the *closure* field. Closure is a logical attribute indicating that interpretation of the input has become fixed. This is similar to the kind of closure that typicaly happens when the user hits "return" in a traditional textual interaction -

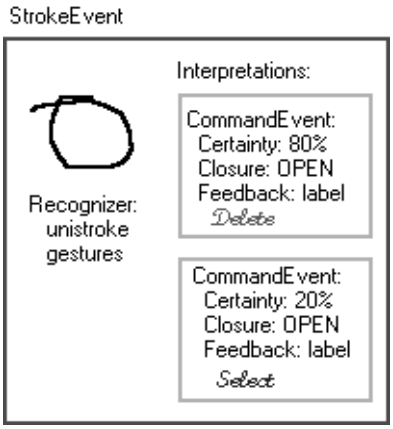

Figure 4: a stroke event for which a unistroke gesture recognizer has returned two interpretations (delete and select).

after that point the user is no longer free to use the backspace key to modify a line of input. In our system closure indicates that a particular interpretation for an ambiguous event is correct from the user's perspective. Interpretations are created with a closure value of *OPEN*. At this point, the application does not yet act on the interpretation, although it does provide feedback for it. When the user selects an interpretation, its closure value is changed to *CLOSED*, and the application can act on it. Essentially, closure indicates to the application how safe it is to take some (potentially irreversible) action based on an interpretation.

Finally, each interpretation has a feedback interactor which can be used to display it on screen. For example, the alternatives in Figure 5a use a line interactor, and the alternatives in Figure 5b&c use a normal menu label interactor. These interactors can be subclasses of any standard GUI interactor as long as they implement a simple interface which allows the *MAD* to request their source event and a text representation for situations where graphics are not tenable. This means the application designer can take advantage of the existing interactor hierarchy instead of having to replicate parts of it.

Event Life-Cycles

As discussed above, our ambiguous event handling architecture delivers information about changes to events as well as the creation of events. We provide several different event dispatchers for delivering information about event changes to interested parties. An event dispatcher is a class which uses state information to decide where and when to deliver events. Events may be delivered to interactors, recognizers, the application, or any other component which has an interest in them. Dispatchers simplify the job of the component receiving the event by keeping track of state and only delivering events to the component when they are useful.

For example, many GUI toolkits have *mouse-motion* event dispatchers which deliver information about mouse move and drag events to interested interactors while filtering out all other events such as key presses, mouse clicks, *etc.* Most of these are *positional* dispatchers. Mouse events are only delivered to interactors when their *position* falls inside the interactor's bounding box. Our *stroke* event dispatcher (Figures 2&3) is another example —it delivers events about stroke start, feedback, and end to components under the stroke. It is, therefore, a type of positional dispatcher. Other event dispatchers may be focus-based. For example, a recognizer can subscribe to a focus dispatcher in order to be told about stroke events even when it has no location on screen.

In addition to handling state and filtering events based on type, event dispatchers in our ambiguous event system may only deliver events when a certain type of ambiguous event *change* has happened. For example, our *application_delivery* event dispatcher handles the separation of *feedback* from *action* described earlier. When an ambiguous event is created, it calls *feedback*. When an ambiguous event's *CLOSURE* is changed, it calls *action.* All other changes to events are ignored. This event dispatcher is responsible for hiding the ambiguity from the application.

We will illustrate the ways that different event dispatchers deliver events, and interactors and recognizers change them by walking through the example in Figure 3. First, a *stroke* event dispatcher collects mouse events and delivers them to the application in the form of *stroke_start*, *stroke_feedback* and *stroke_end* method calls. An ambiguous stroke event is created as soon as the stroke starts (when *stroke_start* is called). When *stroke_end* occurs, the stroke event is changed to reflect the fact that it has been completed. The gesture recognizer is automatically told about this change by a focus event dispatcher (1). The application, which is notified about *stroke_end* by the *stroke* event dispatcher, creates a MAD at this time and passes the stroke to it. The *MAD* will receive the interpretations from the event system when they are created, and will handle the selection of any interpretations generated from that stroke. The application doesn't have to inform the *MAD* about anything except which stroke it is responsible for. This is very similar to how it would treat any other GUI interactor: If it created a *Button*¸ it would not have to tell the button about mouse events in order for the *Button* to know when the user clicks on it.

The recognizer changes the stroke event by adding new interpretations to it (2), and the *application_delivery* event dispatcher automatically queries the application for information on how to display feedback for each interpretation via the *feedback* method (3).

The *MAD* has asked the focus event dispatcher to notify it each time any change is made to the stroke that the application told it to handle. Because of this, it is automatically notified that the list of possible interpretations of the stroke has changed (4). Different

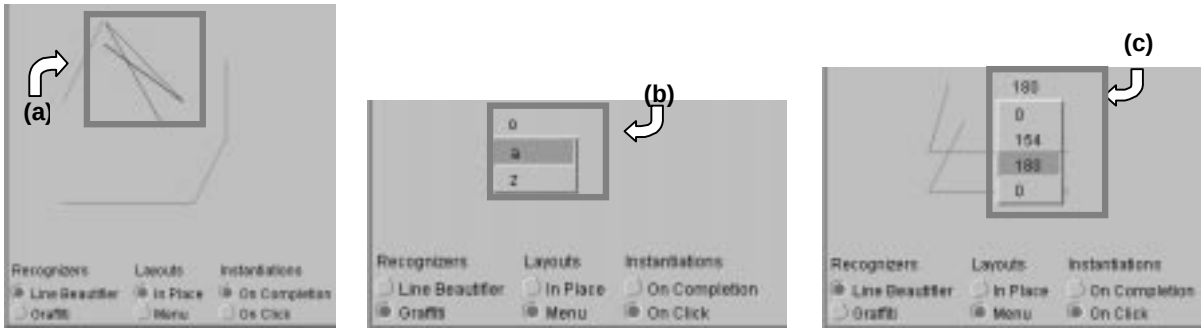

Figure 5: Some example multiple alternatives displays
(a) A multiple alternative display showing 3 lines. The black line is the top choice, the lighter lines are lower choices. The choices are shown in the place they will appear if selected. The *MAD* appears as soon as recognition is completed. The recognizer is a drawing beautifier [12].
(b) A multiple alternative display showing 3 characters. The o is the top choice. The characters are shown in a menu layout, and the menu appears when the user clicks on the top choice. The recognizer is a unistroke gesture recognizer [20].
(c) The same multiple alternative display as (b), this time showing 5 angles. The angles are the text representation of lines generated by the recognizer in (a). The information in the menu indicates a potential line that is described by its degree rotation counterclockwise from the positive x-axis.

*MAD*s will deal with this information in different ways, but they all have the goal of resolving the ambiguity with the user's help. For example, a *menu MAD* such as that in Figure 5b will only display the top choice until the user clicks on it. The user can then drag down and release over the correct interpretation. These mouse events are delivered by the normal event dispatcher just like any other non-ambiguous events received by the application or the *MAD*. Once the user has selected an alternative, the *MAD* notifies the event handling architecture that the interpretation's closure has changed to *CLOSED* (5). By default, the *MAD* also removes itself from the application's GUI.

The *application_delivery* event dispatcher sees that an interpretation's closure has been changed and calls the applications *action* method (6), passing that interpretation to the application. Since the *action* method tells the application exactly which interpretation has been selected, the application doesn't need to do any additional work to figure that out. In our example, *select* was chosen, so the application will now select the object under the stroke. If the chosen event has a graphical interpretation, the application will create conventional (single interpretation) interactor for it at this point.

EVALUATION OF THE ARCHITECTURE

As the example described above illustrates, this architecture supports our stated goal of hiding ambiguity as much as possible from the application while making it accessible to *MAD*s and recognizers.

In addition, the example in Figure 3 illustrates how our architecture supports the two required extensions to standard event handling. We deliver ambiguous events (arrows 1, 3, 6). We also deliver information about changes to ambiguous events (arrow 2, 4, 5), and we have modified the model of event consumers by delivering the same event to multiple consumers (arrows 3, 4).

Applications, recognizers, or interactors that take advantage of the flexibility of our infrastructure can do far more than the simple separation of *feedback* from *action* provided by default. For example, a recognizer which makes predictions may generate interpretations for a partially completed stroke, and then update those interpretations each time new information arrives. After each update, the MAD displaying the predicted interpretations will automatically be notified.

We chose to use the subArctic toolkit [6, 10], because it was a simple matter to extend subArctic's event handling infrastructure (which was derived from that of the earlier Artkit toolkit described fully in [9]) to support ambiguous event handling. This is because subArctic effectively separates the abstractions for dealing with event handling from the rest of the toolkit. In addition, many of the events we are dealing with are non-rectangular, and subArctic makes it fairly easy to design non-rectangular interactors for representing these events. For example, if the lines in Figure 5a were rectangular, many mouse clicks would overlap several different lines making it hard for the user to select one particular interpretation.

Building MADs

In addition to the benefits described above, our architecture makes it a simple matter to implement and test a variety of *MAD*s. Although we have not yet completely populated the space of *MAD*s, we have implemented two different layout policies, two different

instantiation policies, and two different selection policies; and we have connected two different recognizers up to the ambiguous event handling infrastructure. Figure 5 demonstrates this variety.

In Figure 5a, we show a multiple alternative display in which each alternative is shown in place (where it will be if selected). The *MAD* is instantiated when the stroke is completed. Selection is performed by clicking on a line. The recognizer used is an interactive drawing beautifier [13].

In Figure 5b, we show a *MAD* in which the alternatives are layed out in a menu. The *MAD* is instantiated when the top choice is clicked on. In order to select a choice, the user drags the mouse down the menu and releasing it over the desired item. The recognizer used recognizes a unistroke alphabet.

Figure 5c shows the same menu *MAD* displaying alternatives generated by the recognizer from Figure 5a. Both *MAD*s will work with either recognizer, and they do not need to be told anything about which recognizer is being used or what type of alternatives they will be displaying. Each of the recognizers is a 3rd-party product for which we wrote a small wrapper. Once the wrapper is written, a single method call is used to subscribe or unsubscribe a recognizer from the ambiguous event handling system.

The in-place *MAD* is our base multiple alternative display class. The menu *MAD* inherits from it. There were only two significant changes necessary to create the menu *MAD*. First, instead of using each interpretation's default feedback object, it requests a text feedback object from them (the ability to handle this request is required of every feedback object in our system because some situations may only be able to handle text). Second, we substitute a subArctic menu for the base layout object used by default. We did not have to make any changes at all to the menu class. The menu *MAD* simply implements a subArctic interface through which the menu notifies it when a menu item is selected.

Current Limitations

Although the design of our base *MAD* class makes it easy to switch between recognizers and layout types, it is not as simple to change the instantiation type or selection method. This does not prevent designers from using a variety of selection and instantiation types, it merely means that currently each *MAD* demonstrates only one example of each combination. Ideally, a designer should be able to implement, for example, the "on completion" type of instantiation once and use it with every *MAD*.

Another feature missing from our *MAD* class is the ability to filter alternatives. For example, there are times when the drawing beautification system returns many lines which are very close together. When this happens, the user has very little control over which line they are actually selecting. Ideally, the *MAD* should filter out lines which overlap significantly, possibly providing some other way to access the filtered out choices.

This brings up another issue: The correct answer (from the user's perspective) may not be among the alternatives displayed. One solution is to use the *MAD* technique in combination with other techniques such as repetition, or using an alternate (less error-prone) input method for correction.

There are several limitations at the architecture level as well. First of all, it is possible to create an endless loop during event handling. For example, consider a case where two stroke recognizers (A and B) both update their interpretations each time anyone else changes the potential interpretations of a stroke. A adds some interpretations, which causes B to add some, which causes A to re-evaluate and add slightly different ones, which causes B to re-evaluate and…. For now, it is up to the application programmer to make sure these do not happen, although it is possible to work on a cycle-detection algorithm. In order to make this job easier, we provide unique sequence numbers for each event.

Secondly, we need to spend more time populating the set of event dispatchers used to deliver events. In particular, it would be helpful to have separate ones for *MAD*s and for recognizers, simplifying the use of both. For example, the recognizer event dispatcher might use a method called *recognize* when the recognizer's data is available; the *MAD* event dispatcher might use methods *show* and *hide* to indicate when the *MAD* should instantiate itself.

Future Work

In the case of the *MAD* design, there are issues relating to each of the five dimensions discussed in this paper.

Layout: One issue not normally addressed in the layout of multiple alternatives is how to fit them in the surrounding context without obscuring it or making them difficult to read. A complete solution to this requires the support of something like Chang *et al.'s* fluid negotiation architecture [4]. Negotiations between alternatives and between the application and the *MAD* are both necessary.

Instantiation: One interesting problem which we don't currently address is how to discover when automatic selection of an alternative will be erroneous. This affects instantiation because ideally a *MAD* should appear only when the system can't decide which alternative is correct. Currently we show a multiple alternative display any time there is more than one choice. However, there may be situations in which even when the recognizer returns multiple choices it is obvious which one is correct.

Context: The display of context makes it possible to display alternatives without obvious visual

representations, such as commands. Interesting context might include the arguments to a command, or the part of speech assigned to a word. Currently our system depends on the recognizer to provide an object pointing to any relevant context about each alternative. For example, the drawing beautification system provides an object holding the constraints satisfied for each alternative. This has the same problems as making the *MAD* depend on the application for information about alternatives. We need to investigate the possibility of sending context through the event infrastructure just like ambiguous events.

Interaction/Selection: Currently our *MAD*s all use unambiguous events for interaction and selection. Suhm [22] and Huerst, Yang & Waibel [12] have looked at recognition-based methods for error handling, but we only aware of one example which applies recognition-based methods to multiple alternative displays [5]. We plan to investigate new types of *MAD*s which take ambiguous events as input for interaction and selection. For example, the user might select an alternative with a gesture rather than a mouse click.

Feedback: Designers have rarely addressed the issue of how to display multiple alternatives which are commands. Marking menus are one of the few examples of this, and they are limited to a certain type of gesture, displayed only at the user request, and only display commands as names or pictures. For example, predictions of gesture completions may be best displayed as animations, a technique only rarely used to display alternatives [4]. Other ways to display commands include abstract pictures and textual/verbal descriptions. In addition, because commands may not all be associated with one central location on screen, we may need to use color or size to make them more visible.

In terms of our ambiguous event handling architecture, we plan to extend it to handle the full set of error handling issues uncovered in our survey. This will require several additions. First of all, users may change their minds. What should happen when they change which interpretation they have selected? There is extensive research into the undo problem for unambiguous events. It will be interesting to investigate the same problem with regards to ambiguous events. At a minimum, our architecture will need to be extended to keep a history of events and changes to events.

Another issue which we will need to address is the segmentation problem. Segmentation involves deciding which input tokens (e.g., strokes for pen-based input) to group together to send to a recognizer. Segmentation is error prone and segmentations may change over time. Because of this, previous recognition results may be invalidated. Ambiguous events may disappear completely or need to be re-recognized as segmentation changes. We believe that our basic design will scale to handle this problem. When events change, we make information about that change available. A change in segmentation is just another change we will need to deliver to interested parties. Since segmentation is error prone, we will also need to investigate how to present segmentation alternatives to the user.

CONCLUSIONS

This work is based on the claim that uncertianty is inherent in recognition-based technologies, and cannot be eliminated nor ignored. This means that in order to make real use of recognition in the interface, uncertainty needs to be dealt with explicitly, and that the user needs to be involved in at least some cases. To make this a practical reality, it is important that toolkits provide an infrastructure and interactor library which can be reused and extended.

There are two commonly used interface techniques: repetition and interaction with multiple alternatives. We have chosen to focus on the second because its complexity and the need to integrate it with the existing interface make it an interesting, difficult problem. In order to do this, we surveyed existing one-off solutions and developed toolkit-level support for implementing them. Our solution was to extend event handling to include the concept of ambiguous events. From the application's perspective, ambiguous events are almost indistinguishable from unambiguous events. The only change is that the application must separate feedback about ambiguous events from the action taken once the user has selected the correct interpretation. On the other hand, other components using our toolkit have access to a much richer set of information about event creation and changes to events. This allows us to build powerful new interactors. In particular, it allows us to build multiple alternative displays, the error handling technique we originally set out to provide.

ACKNOWLEDGMENTS

The authors thank members of the Future Computig Environments Group in the College of Computing for their comments on early drafts of this work, especially Beth Mynatt, Blair MacIntyre & Anind Dey. The Java-based unistroke gesture recognizer used for demonstration in this paper, GDT, was provided by Chris Long from UC Berkeley as a port of Dean Rubine's original work [20]. Takeo Igarashi provided the drawing beautifier recognizer, Pegasus, that was also used for demonstrational purposes [13]. This work was supported in part by the National Science Foundation under grants IRI-9703384, EIA-9806822, IRI-9500942 and IIS-9800597.

REFERENCES

1. Alm, N., Arnott, J.L. and Newell, A.F. Prediction and conversational momentum in an augmentative communication system. *Communications of the ACM 35*, 5 (1992) 46–57.

2. Apple Computer, Inc. The Newton MessagePad

3. Brennan, S.E. and Hulteen, E.A. Interaction and Feedback in a spoken language system: A theoretical framework. *Knowledge-Based Systems 8*, 2-3 (1995), 143–151.

4. Chang, B., Mackinlay, J.D., Zellweger, P. and Igarashi, T. A negotiation architecture for fluid documents. In *Proceedings of the ACM Symposium on User Interface Software and Technology*. ACM. November, 1998. pp.123–132.

5. DragonDictate product Web page. Available at: http://www.dragonsystems.com/products/dragondictate/index.html.

6. Edwards, W. K., Hudson, S., Rodenstein, R., Smith, I. and Rodrigues, T. Systematic Output Modification in a 2D UI Toolkit. In *Proceedings of the ACM Symposium on User Interface Software and Technology*. ACM. October 1997. pp. 151-158.

7. Goldberg, D. and Goodisman, A. Stylus User Interfaces for Manipulating Text, *in Proceedings of the ACM Symposium on User Interface Software and Technology*. ACM. 1991. pp.127-135.

8. Greenberg, S., Darragh, J.J., Maulsby, D. and Witten, I.H. Predictive Interfaces: what will they think of next? In *Extra-ordinary Human-Computer Interaction: Interfaces for Users with Disabilities*, Cambridge University Press. 1995., pp.103–139.

9. Henry T., Hudson S. and Newell, G. Integrating Snapping and Gesture in a User Interface Toolkit. In *Proceedings of the ACM Symposium on User Interface Software and Technology*. ACM. October 1990. pp. 112-122.

10. Hudson, S. and Smith, I. Supporting Dynamic Downloadable Appearances in an Extensible User Interface Toolkit. In *Proceedings of the ACM Symposium on User Interface Software and Technology*. ACM. October, 1997. pp. 159-168.

11. Hudson, S., Newell, G. Probabilistic State Machines: Dialog Management for Inputs with Uncertainty. In *Proceedings of the ACM Symposium on User Interface Software and Technology*. ACM. November, 1992. pp. 199-208.

12. Huerst, W., Yang, J. and Waibel, A. Interactive error repair for an online handwriting interface. In *Proceedings of CHI'98 Human Factors in Computing Systems.* ACM/SIGCHI. April, 1998. pp. 353–354.

13. Igarashi, T., Matsuoka, S., Kawachiya, S. and Tanaka, H. Interactive beautification: A technique for rapid geometric design. In *Proceedings of the ACM Symposium on User Interface Software and Technology*. ACM. October, 1997. pp.105–114.

14. Kurtenbach, G. and Buxton, W. User learning and performance with marking menus In *Proceedings of CHI'94 Human Factors in Computing Systems*. ACM/SIGCHI, N.Y. 1994. pp. 258–264.

15. Kurtenbach, G., Moran, T.P. and Buxton, W. Contextual animation of gestural commands. *Computer Graphics Forum 13*, 5 (1994), 305–314.

16. Mankoff, J. and Abowd, G.D. Error correction techniques for handwriting, speech, and other ambiguous or error prone systems. Georgia Tech Technical Report, GIT-GVU-99-**. May, 1999.

17. Marx, M. and Schmandt, C. Putting people first: Specifying proper names in speech interfaces. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM, N.Y. November, 1994. pp. 30–37.

18. Netscape Communications Corporation Web page. Available at http://www.netscape.com.

19. Rhodes, B.J. & Starner, T. Remembrance agent: t: A continuously running automated information retrieval system. In *Proceedings of PAAM '96 International Conference on The Practical Application Of Intelligent Agents and Multi Agent Technology*. 1996. pp. 487–495.

20. Rubine, D. Specifying gestures by example. *Computer Graphics 25*, 4 (July 1991), 329–337.

21. Schomaker, L.R.B. User-interface aspects in recognizing connected-cursive handwriting. In Proceedings of the *IEE Colloquium on Handwriting and Pen-based Input*, number 1994/065, The Institution of Electrical Engineers, London, 1994.

22. Suhm, B. Multimodal interactive error recovery for non-conversational speech user interfaces. PhD Thesis, Kalsruhe University, 1998.

23. Sukaviriya, P., Foley, J. and Griffith, T. A second generation user interface design environment: The model and runtime architecture. In *Proceedings of INTERCHI'93* (April 24-29). ACM, N.Y., 1993. pp. 375–382.